Using an architectural approach to integrate heterogeneous, distributed software components

John R. Callahan[1]
Department of Statistics & Computer Science
Concurrent Engineering Research Center
West Virginia University

James M. Purtilo[2]
Computer Science Department
University of Maryland, College Park

Abstract

Many computer programs cannot be easily integrated because their components are distributed and heterogeneous, i.e., they are implemented in diverse programming languages, use different data representation formats, or their runtime environments are incompatible. In many cases, programs are integrated by modifying their components or interposing mechanisms that handle communication and conversion tasks. For example, remote procedure call (RPC) helps integrate heterogeneous, distributed programs. When configuring such programs, however, mechanisms like RPC must be used explicitly by software developers in order to integrate collections of diverse components. Each collection may require a unique integration solution. This paper describes improvements to the concepts of software packaging [1] and some of our experiences in constructing complex software systems from a wide variety of components in different execution environments. Software packaging is a process that automatically determines how to integrate a diverse collection of computer programs based on the types of components involved and the capabilities of available translators and adapters in an environment. Software packaging provides a context that relates such mechanisms to software integration processes and reduces the cost of configuring applications whose components are distributed or implemented in different programming languages. Our software packaging tool subsumes traditional integration tools like UNIX MAKE by providing a rule-based approach to software integration that is independent of execution environments.

[1]Supported by ARPA Grant MDA 972-91-J-102 under the DARPA Initiative for Concurrent Engineering (DICE) program, the National Library of Medicine (NLM) Healthcare Information project, and NASA Grant NAG 5-2129 under the Independent Software Verification and Validation (IV&V) program. Formerly supported by ARPA Software Protoyping Technologies (Prototech) and Domain-Specific Software Architectures (DSSA) projects.

[2]Supported by ARPA Software Protoyping Technologies (Prototech) and Domain-Specific Software Architectures (DSSA) projects.

1 Introduction

Most high-level programming languages provide function and procedure call abstractions in order that software developers can define their own operations and reuse libraries of functions written by other programmers. Use of a function or procedure is *seamless* from the viewpoint of the software developer because a "call" is a language abstraction that is independent of any runtime environment, i.e., the use and definition of a function does not change between environments. Such abstractions make programs more reusable and portable to many types of execution environments regardless of their operating system and hardware characteristics.

Even though functions may be defined in separate components such as files or libraries, tools like compilers and link editors must *integrate* these components. Integration is the step-by-step process of translating and combining software components into new components. For example, a piece of source code can be translated into object code by a compiler and then combined with libraries by a link editor to produce an executable program. The resulting executable program defines a runtime implementation of the application for a specific machine and operating system. The tasks of translating and combining these components into programs may be different for each environment. The developer must invoke the proper tools in their proper sequence to build the application, but the components are the same regardless of the environment. The cost of constructing compilers and link editors for each execution environment is offset by the ability to integrate other software across many environments and amortized over all the programs written in the programming language.

The integration process is more difficult, however, if functions are implemented in different programming languages or as remote services in a distributed system. While function and procedure call abstractions can be implemented by several mechanisms in such situations (e.g., pragmas, pipes, remote procedure call (RPC)), developers must often construct the additional software that uses these mechanisms and provides a "bridge" between components. This additional software, such as remote procedure call stubs, is expensive to develop and unlikely to be reused in other applications. Code generators, such as stub compilers, can be used to produce the additional software automatically, but developers must provide interface specifications in such cases. As in homogeneous environments, developers must invoke the proper tools in order to integrate an application into an executable program, but this process is much more complex in heterogeneous applications.

The problem of integrating heterogeneous programs becomes critical as the need for software reuse grows. By reusing existing programs in new designs, we can significantly reduce development costs, but many data representations and runtime environments are incompatible without the use of "bridge" code. For instance, the United States Department of Defense estimates that most of its 1.4 billion lines of code is used to pass data back and forth between incompatible and inconsistent applications [2]. Much of this "bridge" code is redundant and highly-dependent on each system or execution environment.

This paper describes improvements to software packaging concepts presented in [1] and some of our experiences regarding use of software packaging in constructing complex software systems from a wide variety of components in different execution environments. Software packaging allows software developers to connect programs together abstractly without explicit concern for reconciling implementation differences. It allows developers to explicitly deal with the logical structure, called the *architecture*, of a software system in terms of components and their connections. The software packager then determines automatically which component implementations to use and how to implement the connections in the architecture. It determines whether or not programs can be integrated based on the types of components involved and the available integration tools, e.g., compilers, linkers, and stub generators). If it is possible to integrate the components, then the packager determines which tools are needed, how to apply them, and the proper sequence of their application.

The software packager determines how to integrate software components based on their *implementation types*. Implementation types are based on characteristics like programming language, entry points, and other data and control properties. Such types are independent of the data or object type supported by an implementation. Current integration methods require that developers informally know what types of components are compatible, their interconnections, how to implement the interconnections, and how to integrate the components in each execution environment. For example, file extensions (e.g., `.c`, `.o`, `.exe`) are used to designate crude implementation types. The software packager relies on a richer set of implementation types to help encapsulate components for improved reuse and integration. It determines if components are compatible and how to integrate them based on their types and interconnections.

Software packaging requires only that the developer know about components, called *modules*, and their interconnections in software designs. Each module must have at least one or more associated implementations. The software packager chooses compatible implementations for each component and relies on a set of production rules in each environment to determine how to implement the interconnections and integrate the different types of components. The production rules are unique for each environment and are based on the set of available tools. In most environments, tools are used informally by programmers. Production rules, however, formally characterize the use of tools in integration processes. The cost of constructing the production rules is balanced by the ability to port the applications to other environments and amortized over all applications packaged in the execution environment.

Software packaging reduces the cost of integrating software systems when compared to existing configuration methods such as UNIX MAKE and similar tools. Using these traditional tools, developers must explicitly specify the process of integrating computer programs. If a program is reconfigured but the logical structure is unchanged, then the integration process may be altered. For example, if a component is reconfigured as a set of remote procedures, it will probably require changes to the integration process. Reconfiguring a system includes such tasks as moving a system to another computing environment, distributing components on processing elements, and implementing components in different programming languages. Such changes strongly impact how a system is integrated. The software packager reduces the impact of reconfigurations by providing a high-level approach to integration for a set of programs and processors, much like a compiler does for a single program and machine.

Previous integration techniques focus on integration mechanisms like remote procedure call but they do not address the relationship between the tools and their use in integration processes. With tools alone, the developer must still specify how to integrate an application explicitly. The developer must alternate between abstraction and implementation: connecting components together in a logical structure and then implementing those connections via different integration mechanisms. Software packaging leverages integration tools automatically to derive integration processes based on the architecture of application solutions. Developers deal only with this logical architecture. This results in faster development and allows developers to deal with the architecture of a system in a seamless fashion.

1.1 Integrating a Heterogeneous Application

Suppose our task is to develop a factorial application by integrating existing source components written in different programming languages. The application is modularized into two components: a Driver module for dealing with input/output and a Factorial module that implements a fact function. One solution based on this design is shown in the source code in Figures 1 and 2. This code consists of a Driver component implemented in the FORTRAN programming language and a Factorial component implemented in the C programming language. The Driver component relies on an `ifact` function that is external to its definition. The Factorial component implements a `fact` function. The Driver invocation of `ifact` sends an integer as an argument and expects an integer in return. The Factorial component provides a similar interface except that the names `fact` and `ifact` are not identical.

```
n = 5
nresult = ifact(n)
write(*,*) "The factorial of ",n," is ",nresult
end
```

Figure 1: FORTRAN implementation of the Driver component (in file driver.f)

```
fact(x)
      int x;
{
      if (x <= 1) return 1;
      else return (x * fact(x-1));
}
```

Figure 2: C implementation of the Factorial component (in file factorial.c)

While the two programs are compatible relative to the semantics and arguments of their interfaces, their implementations are incompatible. The major problem is that the FORTRAN and C runtime environments are different in several ways, e.g., they represent strings and numbers differently and have different function call semantics. Such incompatibilities can be reconciled by introducing additional software to bridge the gap between the two implementations and their runtime environments.

There may be several ways to integrate heterogeneous programs in an execution environment. For example, in our environment we can integrate the FORTRAN and C implementations by first translating the FORTRAN code into C using the *f2c* tool [3]. In addition, we must adapt the resulting code with a *wrapper* because the arguments to `ifact` are called by reference as a result of the *f2c* translation. A wrapper is additional code that is used to transform, convert, and bridge runtime differences and data representations. In this case, the function call to `ifact` is implemented by a procedure call mechanism that operates through the wrapper needed to link to the `fact` function.

Figure 4 illustrates the steps necessary to integrate the components of our factorial program. The process is shown as a partial order to reflect the dependencies between integration steps. Assume the FORTRAN and C code shown above are stored in the files `driver.f` and `factorial.c` respectively. First, the FORTRAN code is translated in C by the *f2c* tool and stored in the file `driver.c` (a). Next, the wrapper code must be supplied. In most cases, wrappers are written by the developer or generated from specifications written in an interface definition language. We will deal with wrapper generation in a later section. In this case, the wrapper file `wrapper.c` is supplied and contains the code

```
int
ifact_(x)
      int x;
{
      return fact(*x);
}
```

that describes a C function `ifact_` that invokes the `fact` function with a value parameter. This code is needed to adapt the call from the translated FORTRAN code. The suffix "_" on the name of the wrapper function is necessary because the translator adds this for all FORTRAN function calls. This wrapper also handles the naming conflict between the use of `ifact` and the definition of the `fact` function name. Software packaging views wrapper production as a form of backpatching in heterogeneous, distributed environments and relies on separate tools to produce such code.

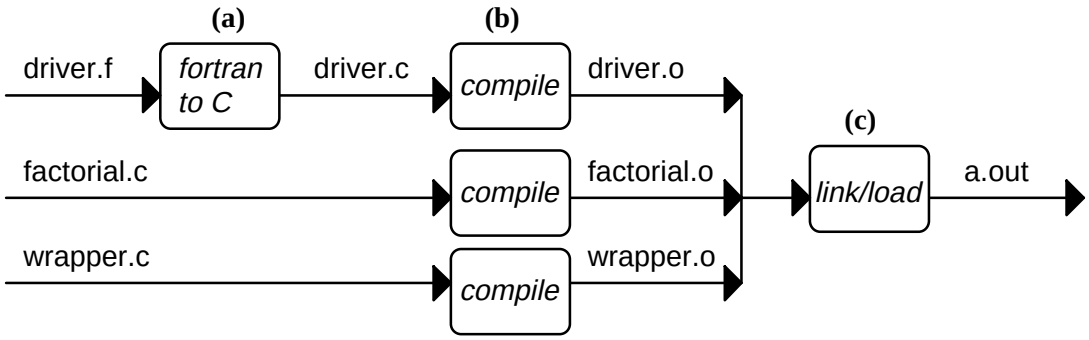


Figure 3: Integrating FORTRAN and C implementations

Next, the files `driver.c`, `factorial.c`, and `wrapper.c` are then compiled to produce the object files `driver.o`, `factorial.o`, and `wrapper.o` (b). Finally, the object code is linked together with the FORTRAN and C run-time libraries to produce an executable program in the file `a.out` (c). The user can then run the program to produce the desired output:

```
% a.out
The factorial of 5 is 120.
%
```

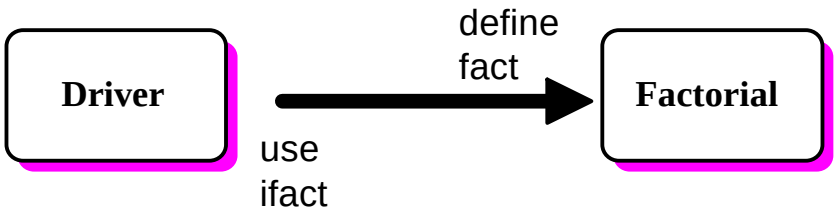
This process represents one possible way of integrating the application into an executable program. Depending on the tools in an environment, there may be many different alternatives. We have used software packaging in this fashion to integrate many different types of applications consisting of heterogeneous components including C, C++, FORTRAN, Lisp, ISETL, Tk/Tcl and Ada.

2 Software Packaging

The behavior of any solution to the factorial problem should be independent of how we implement the application and each component with an application. The integration process, however, is highly dependent on what implementations for software components are available. While each component relies on functional abstraction to isolate itself from such decisions, different implementations of each component will influence how the function call abstraction is implemented. In homogeneous programming environments, call abstractions permit seamless integration between components because the abstractions are an integral part of the language and runtime environment. In heterogeneous environments, the software packager provides the same transparency for multiple-language and distributed applications by determining how to integrate diverse components based on the types of components and the available integration tools.

The software packager accepts modular descriptions of applications as input and generates a package that implements the given system in an execution environment. For example, the module structure of the factorial program and its specification in the PACKAGE language is shown in Figure 4. We describe the details of the PACKAGE specification language in Section 6. Given a description of the modular structure of an application and the implementations for each component, the packager generates a package that includes the code, wrappers, and integration steps necessary to build an implementation of the application. The software packager adapts components, chooses compatible implementations, and selects the appropriate tools needed to integrate a system of components.

Given the PACKAGE specification of the architecture of a system including all possible implementations for all modules, the software packager determines the steps necessary to integrate an application in an execution environment. Each environment provides production rules that characterize the abstract




```
include stdpkg.pkg
module Driver {
       use ifact(int)(int);
}
module Factorial {
       def fact(int)(int);
}
implement Root as {
       Driver:    x;
       Factorial: y;
       bind x'ifact to y'fact;
}
implement Driver with fmain {
       FILE=driver.f
}
implement Factorial with cfunc {
       FILE=factorial.c
}
```

Figure 4: Modular architecture and PACKAGE specification of the factorial solution

integrations the are possible in that environment. These rules are reused by many applications in an environment. The developer supplies only the modular description of a program and implementations of the components. The packager uses the production rules in each environment to determine how to integrate these implementations based on their types and interconnections.

The major advantage of software packaging lies in its ability to determine the integration steps automatically for software products after reconfigurations. Existing methods require extensive changes to configuration programs (e.g., MAKEFILES) after application components are added, reimplemented, or distributed. In a later section, we compare software packaging with existing methods, UNIX MAKE and remote procedure call, to show that reconfigurations are more easily accommodated through the use of PACKAGE specifications.

3 Problems

There are many reasons why differences exist between software components that make integration difficult. Here, we outline the reasons why interconnections between heterogeneous systems are desirable but difficult to achieve because of their implementation differences. Our solution to these problems is motivated primarily by the economic need to reuse existing software in new systems despite their implementation differences. A high-level, modular software design can be reused in different contexts regardless of how its components and their connections are implemented. With software packaging, an application can be integrated using several different technologies. Software packaging does not dictate a single integration mechanism (e.g., RPC), but uses existing technologies to implement interconnections in architectures in order that developers can focus on how to structure their application instead of how to interconnect the components in a variety of environments.

Legacy Code and Systems. Old systems must coexist with new systems because it would cost too much in many cases to upgrade all components in a collection simultaneously. An existing program cannot be discarded simply because another program is installed. New programs must be compatible with existing systems or the new methods must be adopted incrementally by adapting the old system or gradually converting it. Furthermore, existing databases and files may not be compatible with new programs. For these reasons, many programs are designed to be backward-compatible to avoid isolating existing users. Even when a new system is introduced, old systems may remain for long periods of time until they are upgraded. In many cases, service must be provided continuously even while upgrades are in progress.

Coupling between Software Components. Besides data representation conflicts, computer programs also differ because they depend on different execution contexts. For example, a program that uses one set of interrupt signals to control its execution cannot be combined in the same address space with a program that uses the same signals for other reasons. In this case, the programs must execute in separate address spaces with their own interrupt vectors. Such programs are coupled to specific execution contexts external to their implementation, i.e., they depend on specific runtime environments. By definition, coupling reduces software reuse in other contexts. Coupled programs can be adapted to new contexts (e.g., SunView programs can run under X windows with minor source code changes and the use of the XView library), but this is rare and expensive to implement. It is, however, an alternative to reimplementation.

Specialization of Languages and Systems. Computers, languages, and protocols are specialized for problem domains. Numeric problems may best be solved in FORTRAN rather than LISP. We must recognize that computer systems, languages, and protocols will continue to be specialized. Specialization is necessary because it allows developers to construct solutions in terms convenient to a specific problem domain. However, problems decompose into subproblems in several domains. Designing an airfoil, for example, may require a computer-aided design (CAD) program as well as a finite-element processing program. We must allow developers to use the tools that are most appropriate to their problem domains and find ways to integrate the diverse solutions to their subproblems.

Data Management. Certain representations of information may be accessed faster than others. A list of names and phone numbers may be alphabetized for quick access by humans, but an operator may need a list ordered by phone numbers. People and operators require different "views" of the same information. Many programs store information in different formats to access their view of the data, but this limits exchanging the information with other programs that have different views. The information can exist as copies in separate views, but keeping the separate copies consistent if changes occur is a difficult problem. Developers must balance the tradeoffs between efficiency and consistency control when designing a system.

Efficiency of Execution. Even though complete portability of source code is desirable, it is unrealistic given that different hardware and operating system platforms offer different performance and that different solutions to a problem may need to tradeoff performance for other factors (e.g., space, functionality). Thus, different implementations must exist that have different performance characteristics due to the tradeoffs inherent in the overall system design.

Distribution of Components. The trend in modern computer systems is toward decentralization. Users now have powerful processing capabilities at their personal disposal at reasonable prices. The lack of central control over computing resources, however, has resulted in the development of divergent systems with their own languages and protocols. The existence of legacy systems and the need for efficiency and specialization has created enormous differences between systems. Distributed computer systems can be connected via many technologies and this choice impacts the performance and reliability of the final system. The location and access to information services in such systems are important decisions for designers who must accommodate these configuration constraints.

Software packaging allows developers to combine software components with different implementations. Although such differences *must* be reconciled in some fashion in order to integrate the components, the software packager determines what bridges are needed based on the types of components. For each execution environment, a set of production rules describe the types of integrations possible. Thus, it is possible to integrate components only if the proper tools exist. Given the proper tools, the details of integrations are hidden from the developer.

4 Concepts

Despite the differences between software systems, many programs can be integrated if the proper tools are available. Each step of the integration process involves the use of tools like compilers, linkers, converters, wrapper and stub generators. Such tools are used to integrate a given set of software components (e.g. source files) into a final product (e.g., an executable file). The tools in an execution environment define the types of legal integration processes in that environment. Software packaging allows developers to determine the integration processes automatically based on the types of components in an application.

Like the factorial program, many software applications have a logical structure of components that is independent of how each component is implemented and how the application is integrated. For example, Figure 5 depicts a hammock graph called a *production graph* for the factorial program given the FORTRAN and C implementations for each component. The left side of the hammock graph depicts the application as a composition of Driver and Factorial modules. This side of a production graph is called the *software structure graph*. Implementations are shown as descendants of a module (shown as rounded rectangles). The entire application itself is characterized as a module, called the Root module, as well as its components. At the leaf nodes of the structure graph are primitive implementations (e.g., source code). These represent implementations that cannot be further subdivided. The right side of the hammock graph depicts the integration of the implementations into an executable program. This side is called the *software manufacture graph*. Each node in the manufacture graph represents a translation or combination of components using the tools available in an environment.

Given the software structure graph as specified in the PACKAGE language, the software packager automatically constructs the software manufacture graph. The software structure graph is specified in a textual specification language called the PACKAGE specification language by the developer as input to the software packager. The packager then determines the manufacture graph (if possible) and produces a "program" that builds the application from selected primitive implementations of program components. The program, actually a UNIX MAKEFILE, is unique to each target execution environment.

The packager determines the software manufacture graph based on production rules that characterize the *abstract* software manufacture graphs in an environment. The approach is similar to attribute grammars for a programming language except that the packager uses an inferencing algorithm on the production rules to resolve which implementations are compatible and how to build the target object. Although the rules are complex to construct, they are intended to package many applications in a single environment.

5 Software Structures

Regardless of how the factorial application is integrated, its architecture remains the same as shown in Figure 4. Programmers who must maintain software products implicitly refer to the architecture to orient themselves with the layout of a product. This section describes a technique for specifying such structures which are explicit and independent of particular execution environments.

One can specify a computer program in many forms. Most programs are comprised of files, statements, functions, variables, and other components. These components depend on each other and their execution is sequenced in some fashion so that the overall program has the desired behavior. Each component relies

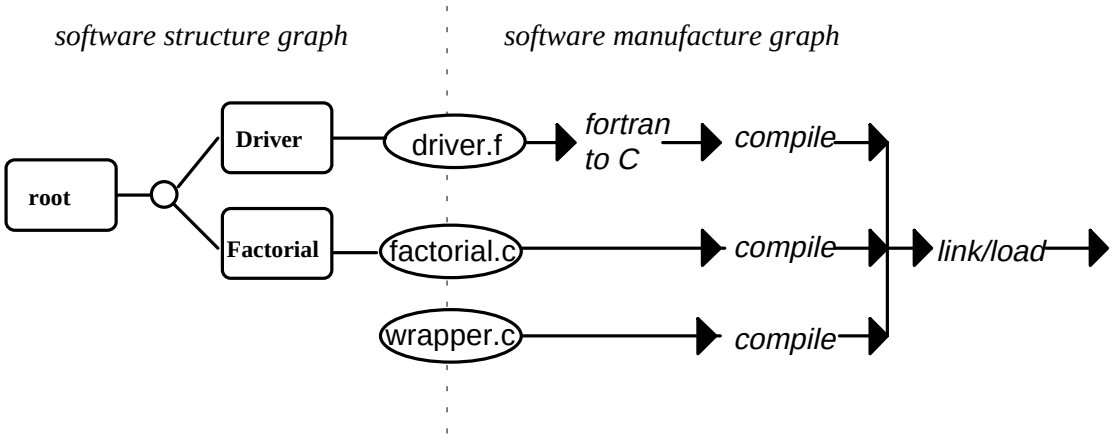


Figure 5: A software production graph for the factorial application

on resources defined by other components. The structure of an application includes a description of each component and the dependencies between them.

One major problem is how to specify a component in a manner that is independent of any implementation [5] [6] [7]. Often, it is impossible to completely separate specification and implementation from a system perspective. We present an approach to describing software structures, called *structure graphs*, that can be used to describe many levels of design: from gross structures to statement-level constructs. Software structure graphs are based on MILs [8], but differ from previous efforts in that our structures are hierarchical, components may have multiple implementations, and attributes and constraints are managed in software structure graphs much like in attribute grammars. This allows for selection of compatible implementations based on their types and other properties.

5.1 Modules

We view any software artifact as a module --- a black-box characterized only by the specified behavior of its interface. An interface is a collection of ports or channels on which messages are received or generated. Ports represent resources implemented by a module: function calls, events, or input-output streams. Within a module, but hidden from the outside, is an implementation. A module may have several implementations but only one may be "inside" the module at a time. For example, a program with the same input-output behavior can be written in two different programming languages. From the outside, it does not matter which implementation is chosen, but its behavior should be consistent with its interface specification.

5.2 Choice

Portable software products often have multiple implementations of their components to handle special cases, i.e., different device drivers may be configured depending on the target platform. In the worst case, a different implementation of the entire program must be build for each target platform. Choosing the appropriate implementations depends on the target platform. Differences between component implementations can be large or small. A single source file may represent multiple implementations because it may be compiled differently depending on the target platform. For example, the `#ifdef` macro in C is used frequently to compile alternative parts of source code depending on a configuration context.

Choice is a fundamental constructor in large software structures. Even a simple data abstraction may have multiple implementations. For example, the Map data type (i.e., associative arrays) in the GNU C++ library has the following implementations:

AVLMap	implement maps via threaded AVL trees
RAVLMap	implement as AVL trees with ranking
SplayMap	implement maps via splay trees
VHMap	implement maps via hash tables
CHMap	implement maps via chained hash tables

These implementations are specified as subclasses of the class Map, but their interfaces are identical. Such relationships between abstract classes like Map and its subclasses occur in object-oriented systems that do not separate subtyping and subclassing [9]. A subtype refines an interface whereas a subclass represents an alternative implementation. Module-oriented programming distinguishes between the two concepts by separating interfaces and implementations and providing for implementation choices within software structures.

The choice of an implementation for any module is based on a variety of factors including the required performance of operations, storage overhead, and data representation strategy. An implementation chosen for one part of a system may constrain implementation choices in other parts of the system. Implementations are compatible relative to such constraints. For example, choice of an implementation using dynamically allocation may mean that all components must use the same memory management style.

5.3 Composition

Another fundamental constructor in software structures is composition. Groups of modules are connected together because they define and use shared resources. For example, the factorial solution in Figure 4 is a composition of two module instances: an instance of the Driver module and an instance of the Factorial module. Composition is an implicit operation in most programming systems. Most programming languages bind uses and definitions of resources together if their names are the same (*by-name* binding). Use of a resource implies that the component defining the resource must be integrated into the product at some point. Linker/loaders assimilate components by matching uses to definitions. Our use of explicit module interfaces and bindings is necessary in cases where integration requires more complex bridges between components. This is particularly true in heterogeneous, distributed systems where remote procedure call (RPC) stubs or other types of links must be generated to integrate components at runtime. Strict encapsulation permits our packaging system to wrap components in new contexts as needed.

5.4 Software Structure Graphs

We combine choice and composition within a framework for constructing descriptions of software products called *software structure graphs* or simply "structure graphs." A structure graph is a directed (possibly cyclic) graph whose root represents a software product and its alternative implementations at many levels. There are two types of module implementations within a structure graph: composite implementations and primitive implementations. Within a graph, alternatives represent subsystem implementations. At the leaves of the graph are primitive implementations (e.g., source code, programs, services, etc.). A structure graph is similar to an AND-OR graph. The children of the root module represent implementation alternatives (OR nodes). Each alternative is either a composite implementation (AND node) or a primitive implementation (P node). Thus, a structure graph is a hierarchical description of an application, its subsystems, and alternate implementations. A structure graph is not a tree because there may be sharing at levels of the graph and cycles involving recursive implementations (i.e., a subsystem implementing a Factorial module may itself include an instance of itself).

The leaf nodes of a structure graph are called "primitive" because they correspond to native implementations of modules in an environment that cannot be broken down further into subsystems. Typically, primitive implementations correspond to source code files, but may also represent services, tools, data, or any software artifact or collection of artifacts. Software structure graphs do not limit the developer to one-to-one mapping to files, rather multiple files could be associated with a single primitive implementation or a single file could be associated with multiple primitive implementations.

Figure 6 represents a structure graph for the factorial example in which the server has an additional implementation: a remote service. Rectangles represent module instances, open circles represent compositions and ellipses represent primitive implementations. In this case, each module has only one instance in the structure graph. In general, a module's implementation subgraph is copied below wherever an instance of the module occurs. The structure graph represents all alternatives for components within the application. In the factorial example, the Driver module has one primitive implementation --- the FORTRAN implementation. In Figure 6, the Factorial module has two possible implementations: the C implementation and the remote implementation.

The structure graph provides developers with an explicit model for constructing, viewing, and maintaining alternative versions of software systems and their implementations. All software components are "black boxes" that may have one or more implementations. The software packager uses environment-specific rules to choose compatible implementations for components within a structure graph.

The software packaging specification language is a *module interconnection language* (MIL) that allows programmers to describe software structures in terms of choices and compositions of software modules. A module specification describes the resources provided and used by a software component. This is more general that an object-oriented approach that describes software components only in terms of resources they provide. For example, the interface of a stack *object* is typically described as providing three basic functions: *push*, *pop*, *top*. Figure **7** depicts a generic stack module with the same interface, but we associate two implementations with the stack module. These implementations are composite implementations consisting of other module instances. The external ports of the stack module are connected to ports of internal modules. This technique, called *aliasing*, represents the bridges between higher and lower level abstractions in the architecture of a system. For example, a composite implementation includes instances of ArrayStack and Array modules that implement stacks using arrays while the other implementation includes instances of ListStack and List modules that implement stacks using linked lists. The graph for the stack module and its implementations are shown in Figure **8**.

6 The Package Specification Language

This section is designed to be reference manual that describes the syntactic units for the PACKAGE specification language. The PACKAGE language is used to describe software structure graphs for applications. The PACKAGE language is a module interconnection language (MIL) in which software components are described as units (either processes or static code) that provide and use resources. An application contains instantiations of modules and connections between resources uses and definitions.

A PACKAGE specification describes the software structure graph for an application. A specification describes a directed, rooted graph (possibly cyclic) whose root node represents the entire application. Each PACKAGE specification *must* describe at least one implementation for the application either as of a collection of components (a composite implementation) or a single object (a primitive implementation). The name `Root` is a distinguished lexical identifier within a PACKAGE specification. At least one implementation for the `Root` must be expressed in a PACKAGE specification. Each direct descendent of the root node represents an alternative implementation of the application itself. The structure graph is elaborated by describing the subcomponents, their implementations, and connections within an application.

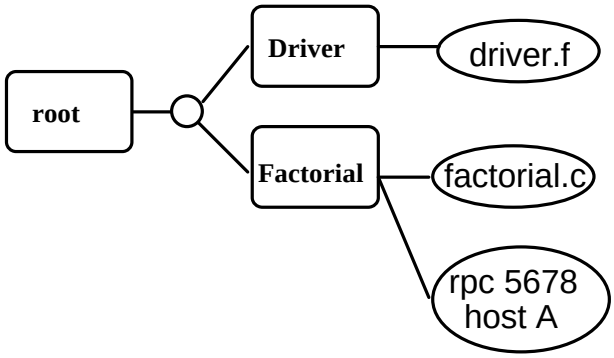


Figure 6: A software structure graph for the factorial product

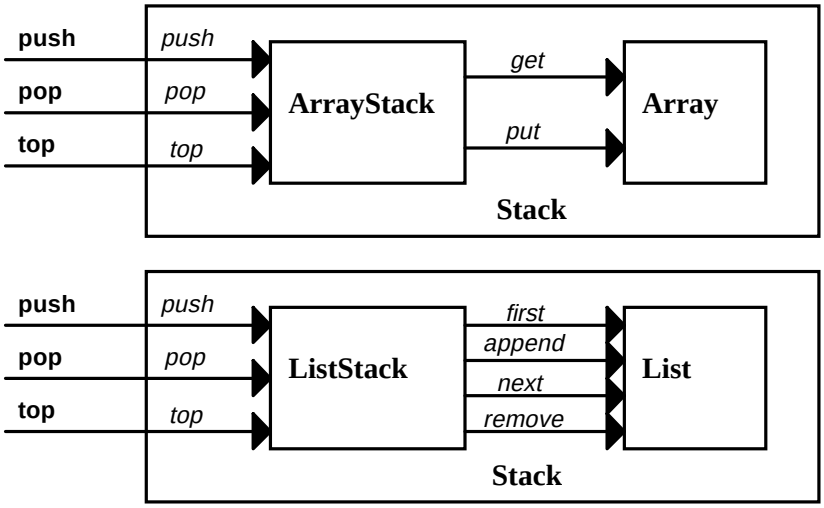


Figure 7: Two alternative composite implementations of a stack module

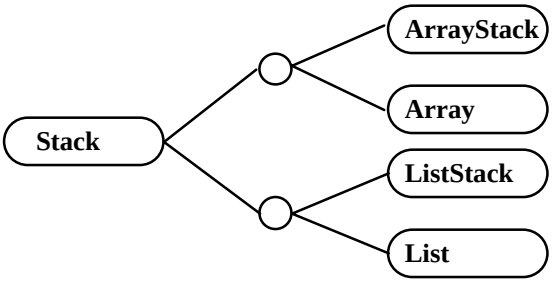


Figure 8: A structure graph showing two alternative composite implementations of a stack module

```
include stdpkg.pkg;
module Driver {
      use ifact(int)(int);
}
module Factorial {
      def fact(int)(int);
}
implement Root as {
      Driver:    x;
      Factorial: y;
      bind x'ifact to y'fact;
}
implement Driver with fmain {
      FILE=driver.f
}
implement Factorial with cfunc {
      FILE=factorial.c
}
implement Factorial with rpcsvc {
      NUMBER=5678
      HOST=A
}
```

Figure 9: A Package specification for the factorial problem with two possible implementations of the Factorial module

Figure 9 shows a PACKAGE specification that describes the factorial application structure graph consisting of instances of the Driver and Factorial modules. The Factorial module has two associated implementations --- one in C and the other is a remote service. The corresponding software structure graph is shown in Figure 6. The specification is comprised of six syntactic units declared at a global level: a composite implementation for the `Root` module, an interface description for a Driver module, an interface description for a Factorial module, a single primitive implementation for the Driver module and two primitive implementations for the Factorial module. PACKAGE specifications typically consist of a series of declarations of modules and their implementations.

Modules may have multiple associated implementations that are either composite or primitive. Unlike INTERCOL [27] and the previous version of the software packager [1], instances of modules within composite implementations do not need to specify which implementation should be used. The choice is determined by the packager tool. The PACKAGE specification enumerates *all* possible implementations of modules within an application as a subgraph of a module instance. Tools like NMAKE rely on the UNIX file system to specify choices of implementations in a similar fashion, but a PACKAGE specification explicitly describes this structure without attaching choices to particular file systems.

6.1 Modules

The Driver and Factorial module interfaces are declared before the Root composite implementation in Figure 9 but they could be declared anywhere in the specification. The packager uses a two-pass approach to build the software structure graph from these specifications so that modules can be instantiated before their declaration. The Driver module interface consists of a single "use" port representing a function call to an external resource named `ifact`. The Factorial module interface consists of a single "def" port providing a function resource named `fact`.

While a module may have multiple implementations, it may only have one interface description. Modules are uniquely identified by their name and parameter types. A module declaration is of the form

module ***identifier*** *(parameters) : ancestors {*
ports
attributes
}

where ***identifier***[3] is the name of the module type, *parameters* is a comma-separated list of variable names and *ancestors* is a comma-separated list of module names. Module parameters may be referenced within the body of the module specification. They expand to string or numeric values in the same manner as attribute references. Module ancestors refer to modules from which a module inherits attributes and ports in a fashion similar to inheritance in object-oriented languages except this style of inheritance is involves only the interface not the implementations of the ancestor modules.

Ports within a module are distinguish by their names and parameter types. Port names may be overloaded with different parameter types. Ports and attributes are explained in a later sections. If a module has no ports or attributes, it may be declared as

module ***identifier*** *(parameters)* ;

The Root module is the only module not specified in a package. It has the implicit declaration

```
module Root;
```

that is predeclared in the standard package header which is included via the `include` directive. The Root module may have default ports and attributes, but these are usually specific to an execution environment.

6.2 Ports

Ports are associated with modules by instantiating them within a module declaration. In our example, the Driver module has a single "use" port and the Factorial module has a single "def" port. A "use" port represents a function call that expects a single value in return. A "def" port represents a function implementation. There may be many types of ports including sources, sinks, and errors. A port instance within a module declaration is of the form

(min, max) **porttype** ***identifier*** *(path1) ... (pathn) : attributes* **;**

where ***identifier*** is a `use`, `def`, `src`, `snk`, `err` or any other user-defined port type. The *min* and *max* are the minimum and maximum number of connectors that may be attached to this port. The minimum and maximum values are used to place constraints on connections to and from a port. For example, a "use" port can only be connected once since multiple connections would imply a broadcast procedure call. Ports of type "def", however, may have an unlimited number of incoming connections corresponding to procedure invocations. The *path* specifications of a port declaration describe the types of messages on the resource. A path is a sequence of message types that include primitive types (integers, strings, floats), other module names, or static data structures known as classes.

Module ports may also have associated attributes. This allows the assignment of arbitrary string and numeric values to keyed names on ports. For example, some work has been done to explore the association of semantic information with module ports. This information is later used to check for port compatibility during the packaging process using external tools. The packager itself is not responsible for

---

[3]Required syntax is show in boldface. Non-terminals are shown in italics. Thus, optional syntax is shown in non-bold and non-bold italics.

determining the syntactic or semantic compatibility of interface port bindings, but rather it invokes external tools as part of the packaging and integration processes.

Data type names in module port declarations (e.g., `int`, `real`, `str`, etc.) are not part of the PACKAGE specification language. Port compatibility decisions regarding type checking, semantics, and other integration checks are made during the packaging process by external tools. The software packager is a framework for organizing such work, but does not dictate any data type names or representation formats.

6.3 Port Types

Port types are declared at the global level. A port type is a pattern specifying a legal port type. Default port types are defined in the standard package header (`stdpkg.pkg`). In general a port type declaration is of the form

> **`port`** ***identifier*** *(argument_type) ... (argument_type) {*
> *attributes*
> *}*

For example, the "use" and "def" port types are declared in the standard package header as

```
port use(...)(?);
port def(...)(?);
```

where the ellipses imply that both port types take any number and types of arguments. The question mark means that both return only one value of any type as a result. Both ports associated with the Driver and Factorial modules in our example are legal instances of their port types.

6.4 Composite Implementations

Modules may have multiple associated implementations. There are two types of implementations: composites and primitives. A composite implementation describes a collection of module instances and their connections. The implementation of Root in our example is a composite implementation. A composite implementation is a circuit-board diagram of connected modules: it describes the subcomponents and the "wiring" between their ports that comprise an implementation for a module. It represents a subsystem because the instances within a composite also have associated implementations, but these are not visible at this level of abstraction. Each instance is simply a black box. In general, a composite implementation is of the form

> *name:* **`implement`** ***identifier*** *(formals)* **`as`** *ancestors* **{**
> *instances*
> *connections*
> **}**

where ***identifier*** is the name of the module type (formal parameters must match as well), an *instance* is a declaration of a module instance and a *connection* is a link between a single port or groups of ports. The connections within a composite implementation describe how to wire the ports of the module instances together. The designer is not constrained to wire ports one-by-one. There are constructs for performing connections by pattern and port type. The form *ancestors* is a comma-separated list of implementation names. A composite implementation may include the instances and connections from other named composite implementations. This allows for the inheritance of composite implementations separate from the subtyping of module interface descriptions.

Instances within composite implementations are specified with an instance name and may have attached attributes. The form of an instance description is

modulename* : *identifier1 *dimensions: attributes , ..., **identifiern*** *dimensions : attributes* **;**

where ***modulename*** is the name of a module and each ***identifier*** is the assigned name of an instance. All module names must be defined, even if a module has a null interface (e.g., Root). Arrays of modules may be declared with multiple dimensions. Attributes can be associated with individual instances. These attribute assignments are scoped on the subgraph below the instance, not the entire composite subgraph.

In our example specification in Figure 9, the Root implementation is the only composite implementation. It contains two instances: one instance of the Driver module and another instance of the Factorial module. Within the Root implementation, the instances of the Driver and Factorial modules are assigned the names "x" and "y" respectively. The `bind` statement

```
bind x'ifact to y'fact;
```

specifies that the `ifact` port of the Driver instance "x" is connected to the `fact` port of the Factorial instance "y". Composite implementations for modules that have ports (unlike Root in this case) may use the bind directive to connect ports of internal instances to the external ports. For example, the specification

```
implement Stack as {
      ArrayStack astack;
      Array a;
      bind 'top to astack'top;
      bind 'push to astack'push;
      bind 'pop to astack'pop;
      bind astack'get to a'get;
      bind astack'put to a'put;
}
```

describes a stack module composite implementation comprised of an ArrayStack instance and an array instance. The alias directives connect inner ports to outer ports within a composite. The specification

```
implement Stack as {
      ArrayStack astack;
      Array a;
      bind *'* to *'$2;
}
```

is equivalent to the previous specification but employs a shorthand notation for connecting groups of ports instead of individually. The connecting phrase

```
bind *'* to *'$2;
```

is a "cliché" for binding ports by name. This phrase is equivalent to name-binding that is employed by many link editors. Connections within packaging specifications are checked to ensure that the connection constraints on ports are within their limits.

Composite implementations of modules represent subclass implementations of embedded module instance. Unlike object-oriented languages like C++ [11], the subtyping and subclassing in PACKAGE specifications are separate. While this has some disadvantages, such as performance, it totally encapsulates software modules and their implementations to promote their reuse in many contexts. The CONIC system [34] takes a similar approach to separate subtyping and subclassing but does not employ multiple

implementations. The RESOLVE programming language [9] employs the same separation and multiple implementations, but implementations are all coded in RESOLVE and distinguished by performance and space characteristics.

6.5 Primitive Implementations

Modules may also have associated primitive implementations. Primitive implementations are different from composite implementations because they refer to native objects that implement a module, not a subsystem of module instances and bindings. In our example, the Driver module is implemented by a `fmain` object and the Factorial module is implemented by `cfunc` and `rpcsvc` objects. The specification of a primitive implementation has the form

`implement` ***identifier*** *(parameters)* **`with`** ***impltype*** **`{`**
    attributes
`}`

where ***identifier*** is a module name and ***impltype*** is an implementation object type. The attributes assign string or numeric values to named variables. Implementation object types are declared in the standard header as

`object` ***identifier*** *: ancestors {*
    attributes
}

where ***identifier*** is the name of the implementation object type (e.g., `fmain`, `cfunc`, `rpcsvc`) and the attributes set default values within primitive implementations of the object type. Object types may also have ancestors that define additional attributes. Object type attributes describe component parameters such as source file names, tools, data files, etc. Details on object types and their attributes are described in a later section.

6.6 Connectors

The `bind...to` operation is not a primitive in the PACKAGE language, rather it is an instance of a {\em connector} type. A connector instantiates an object similar to a module instance. The `bind...to` connector is declared in the standard header as

```
connector bind(use)to(def);
```

Connectors are first-class objects in the Package specification language. Connectors may be primitive or composite. The `bind...to` connector is an example of a primitive connector. In general, connectors may include additional module instances and subconnections to implement a particular connection. The form for a connector type declaration is

`connector` ***opname (srcname) midname (dstname) {***
    instances
    connections
}

where the ***opname*** is the connector operation name (e.g., bind), the ***midname*** is the middle operation name (e.g., to), and ***srcname*** and ***dstname*** are the source and sink port type names (e.g., use, def). A composite connector is similar to a composite implementation and can contain embedded module instances and connections. The instances and connectors are expanded inline into the composite

implementation calling the connector. This means that connectors may only have single implementations and do not form subgraphs in the structure graph.

6.7 Attributes

All syntactic units in PACKAGE specifications, including modules, ports, implementations, and connectors, may have associated attributes. An attribute may be assigned a value that is a string or numeric value, a set or list. For example, the FILE attribute of the Driver primitive implementation in the factorial specification has the string value "driver.f" that specifies the file name of the component. The FILE attribute illustrates the fact that object types need not be associate one-to-one with files, but may have attributes that reference multiple files.

Attributes are dereferenced using the $(*name*) construct similar to that used in tools like MAKE. Unlike MAKE, however, whose attributes are visible at a single lexical level, package attributes are scoped on the software structure graph not on the lexical structure of the PACKAGE specification. This means that an attribute X declared in a Root composite implementation body can be accessed using $(X) in any composite or primitive implementation that becomes a node in the software structure graph below the Root implementation.

Attribute values are evaluated on-demand. This allows attribute values to change during the packaging process as alternative implementations are chosen in the structure graph. Such choices may impact other parts of a configuration. The approach is similar to the use of inherited and synthesized attributes in attribute grammars. Scoping attribute declarations on the structure graph is similar to the use of inherited attributes except that one does not have to explicitly specify the direct inheritance at each level. Likewise, attributes can be synthesized from module instances within composite implementations using the form $X(Y) where X is the name of a module instance and Y is the attribute within X's composite or primitive implementations. If a module instance has multiple implementations, then the value of the synthesized attribute is the concatenation of the remaining viable candidates.

For example, if we added the term `FILE=$y(FILE)` to the composite implementation of the Root module in our factorial package in Figure 9 to yield

```
implement Root as {
      Driver:     x;
      Factorial:  y;
      FILE=$y(FILE);
}
```

then the FILE within the Root implementation would have the value "`factorial.c`" at the onset of the packaging process and a value "" (i.e., the empty string) at the conclusion of the packaging process if the `cfunc` implementation alternative is eliminated because the packager decided to use the `rpcsvc` implementation instead.

In this case, the value of the Root attribute FILE may not be accessed in the declaration of FILE attributes within composite and primitive implementations below the Root because this would introduce a circularity into the evaluation of attribute values within the software structure graph. It is possible to determine the presence of such circularities within attribute grammars and we have adopted this approach in the implementation of the software packager.

6.8 Constraints

Syntactic units within PACKAGE specifications may also contain constraints. Constraints resemble attributes: they are scoped on the structure graph. If an attribute assignment violates a constraint at some higher level in the structure graph, that unit is not expanded. For example, in the PACKAGE specification

```
implement Root as {
      MACHINE == sun
      Driver:     x;
      Factorial:  y;
      bind *'* to *'$2;
}
implement Driver with fmain {
      MACHINE = sun
      FILE = driver_sun.f
}
implement Driver with fmain {
      MACHINE = dec
      FILE = driver_dec.f
}
```

the second primitive Driver module implementation is not expanded as a candidate implementation of the Driver module instance in the Root composite because it violates the higher constraint in the structure graph. The form of a constraint is

*label :: **identifier relop expression***

where the *label* is optional, ***identifier*** is an attribute name, ***relop*** is a relational operator (e.g., ==), and the ***expression*** yields a string, numeric, set, or list value. Constraints may be labeled or unlabeled. Only labeled constraints can be removed. To remove a constraint, the `-NAME` construct removes the last constraint labeled NAME.

The PACKAGE specification language is used to express the form of software structure graphs of software applications. Such graphs represent the implementation alternatives and modular structure of the application. Attributes are visible on the graph and can be used as parameters for lower-level components in the graph. Likewise, constraints can restrict the selection of components included as subgraphs of module instances within embedded composite implementations. Other systems use file system structures to organize software applications and their implementations, but our specification language explicitly states and manipulates this structure.

7 The Software Manufacture

The process of integrating software components is known as a *software manufacture* [13]. A software manufacture is the step-by-step process of synthesizing new software artifacts from existing ones by applying tools available in an execution environment. The available tools include any language translators and integration mechanisms installed in an environment. A software manufacture for an application specifies how to build a product from given set of components. These components include source files, executables, data files, or even executing services.

7.1 Software Manufacture Graphs

In our first solution to the factorial problem, we proposed that the FORTRAN and C components could be integrated via translation and a C wrapper. Based on the directed acyclic graph in Figure 3, one possible process to perform the integration could be

1. Translate the FORTRAN code into C
2. Compile all C source code into object code
3. Link all object code into an executable

since these steps obey the partial order specified by the graph. Any ordering of integration steps that obeys the partial order is valid. Many configuration management tools exploit this partial order that exists within integration processes. A graph that depicts the partial order of steps necessary to integrate software components into a product is called a *software manufacture graph.* The graph specifies the steps necessary to build an application from a given set of components as a partial order. Each node in a graph corresponds to an action that performs a single step in the integration process. A manufacture graph proceeds from left to right with the raw components as input on the left and the final product(s) as output.

7.2 Abstract Software Manufacture Graphs

We introduce the concept of the *abstract software manufacture graph* to characterize the integration processes available in the environment. In any environment, the available tools dictate the types of software manufacture graphs that are *legal*, i.e., those that represent valid integration processes. We characterize the general form of these graphs as abstract software manufacture graphs.

We characterize the abstract form of legal manufacture graphs through the use of production rules. Figure 10 depicts an abstract form of the manufacture graph in Figure 3. In Figure 10, we relabel the nodes in Figure 3 with production rule numbers and the transitions with *object types*. The leaf nodes on the left side of the graph represent primitive object types in the environment. These may or may not correspond to source files and can be associated with other artifacts in the system, e.g., ports, sockets, memory addresses, and services. Object types have associated attributes that identify such properties. For example, an `fmain` is an object type that represents a FORTRAN source file with an execution entry point. The `fmain` object type has a `FILE` attribute associated with it that specifies the source file in the environment. The `fmain` object also specifies an `ENTRY` attribute for the program entry point.

Nodes within the graph are labeled with production rules that correspond to procedures that utilize environment tools. For example, the rule

```
cobj  ⇐    cfunc
```

specifies a method for producing a `cobj` object from a single `cfunc` object. This corresponds to a special case of the "`.c.o`" suffix rule in MAKEFILES. Production rules in abstract software manufacture graphs are similar to those found in attribute grammars: the left side of a rule represents the target while the right side is a list of the components from which the target is constructed. Like symbols within attribute grammars, objects in production rules also have attributes and actions that manipulate these attributes to produce an integration.

Each environment specifies its own unique software integration processes in terms of a set of production rules. The production rules used in Figure 10 are

```
1. exec       ⇐    cfmobj cobjs
2. cfmobj     ⇐    cfmain
3. cfmain     ⇐    fmain
4. cobjs      ⇐    cobj cobjs
5. cobjs      ⇐
6. cobj       ⇐    cfunc
```

These rules form a "grammar" for legal software manufacture graphs in an environment. For example, the graph in Figure 3 is a legal software manufacture graph according to the production rules above.

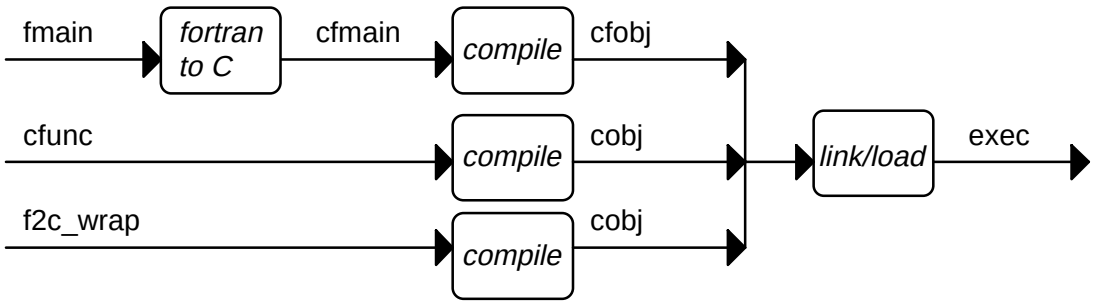


Figure 10: An abstract software manufacture graph for the factorial example

Unlike suffix rules in MAKEFILES or IMAKE procedures, production rules relate the tools available in an environment to integration processes (i.e., sets of related rules). Every environment can characterize its legal manufacture graphs via production rules. New tools are leveraged by adding new rules.

We can derive a *concrete software manufacture graph* given collection of primitive objects and a set of production rules. This is the basic approach of software packaging: determine a means of integrating compatible components based on available integration processes. Developers specify the objects, but they do not specify the production rules. These are written and installed in an environment by system administrators. They are accessed and shared by all developers in an environment. They change when tools are added or removed from the system. Developers must be aware of object types (i.e., leaf node types in the manufacture graph), but this is an improvement over having to remember platform-specific methods as in MAKE or procedures as in IMAKE. In the next section, we explore the specification of objects within application structures that are independent of programming environments.

8 Maps

When the packager selects a particular set of implementations for all modules in a structure graph, the selected leaf nodes (i.e., primitive implementations) comprise an implementation of the entire application. This selection is called a *rendering* of the application in an execution environment. The internal nodes of a structure graph serve only to organize the application choices, constraints, and connections. If we "flatten" a structure graph and map all aliased connections into connections between the leaf nodes, then the connected graph that remains is called the *application graph*. An application graph contains only the selected primitive implementation components and their direct interconnections.

During the integration process, not the packaging process, the direct connections, called *bridges*, between actual components may need to be built between diverse components. Implementing a bridge is similar to backpatching an object file: the link editor connects the uses of functions to their definitions by filling in machine addresses for subroutine jump instructions. In a distributed, heterogeneous application, backpatching is performed by various tools that need information about implementations such as entry points, ports, and other resources used by a component in order to generate wrapper and stub code. Thus, the packager produces an external cross reference file, called a *map*, so that such tools can readily access such information. For example, a stub generator can determine the properties of entry points to a component for which it needs to generate stubs during integration.

In Appendix 2, the MAKEFILE produced by the packager generates the file `wrapper.c` from the file `Map` that is produced by the packager. The program *f2cgen* is a stub generator that produces the wrapper code given the node numbers in the application graph corresponding to the FORTRAN and C primitive implementations. All nodes in the software structure graph are assigned numbers. The stub generator uses these numbers to determine which connections need to be implemented. In this case, the node "5" corresponds to the internal number for the `fmain` implementation and node "7" corresponds to the

`cfunc` implementation. The argument "-d" followed by one or more numbers tells the stub generator to produce code for all connections between nodes 5 and 7 in the `Map` file. This is how the stub code is produced to adapt the call to `ifact` into a call to `fact`.

Production graphs do not explicitly describe the interconnections between components, but at the boundary between the structure and manufacture graph we can derive the application graph that contains all the relevant direct connections. The bindings between interface ports of modules within the structure graph creates a mapping between the ports of primitive nodes at the leaves, i.e. the connections of the application graph. If the packager is able to create a valid manufacture graph, it also produces a map that contains the interconnections between all packaged components. A map is a cross-reference list that can be used by integration tools to construct bridges, like stubs, between components. The map in the factorial example consists of a single connection from the client's factorial port to the server's factorial port. This map can be used by a code generator to produce the `wrapper.c` code. Since stub and wrapper generators rely on the map, it may be included as an object in the production rules that is always available.

A map enables tools like stub generators to access information about and implement the interconnections between components much like a link editor backpatches object code in homogeneous applications. Our approach is extensible because future integration tools can read the maps to determine how to implement interconnections. In our environment, we have built several bridge tools that rely on maps to generate stub specifications from maps for several protocols including Sun RPC [14] and Polylith [15]. We are currently working on stub generators for CORBA [31] and NIDL [17].

9 The Rule Specification Language

The integration rules describe the abstract form of the legal manufacture graphs in an environment, but they do not perform the actual integration. With each rule, we associate actions that contain commands that invoke the proper integration tools. Once the packager determines a valid manufacture graph, the resulting graph is traversed and the actions associated with each node in the graph are executed. The traversal process is similar to the second-pass of a compiler traversing a parse tree built by the first pass.

The rule specification language is described fully in [18] and has changed significantly since [1], but we present a brief description here. A rule specification file is similar to a grammar in Yacc [19]. Each rule specifies a production with a single left-hand-side item and a set of zero or more right-hand-side items. Items on the right-hand-side of a rule may be non-terminals that correspond to other rules or they may correspond to the names of implementation object types (e.g., `fmain`, `cfunc`, `rpcsvc`).

Each rule also has an associated set of actions. Actions are contained within braces {...} and may be interleaved with the objects on the right-hand side of a production rule. An action in the packager is similar to a semantic action in an attribute grammar. For instance, the rule for producing a `cobj` object from a `cfunc` object (rule #6 above) is augmented with an action

```
cobj  :     cfunc
      [ (OBJS) $1.FILE:r ".o " ]
      :{
      %$1.FILE:r.o: $1.FILE
      %     cc -c $1.FILE
      %
      };
```

that produces the output lines

```
factorial.o: factorial.c
      cc -c factorial.c
```

in the resulting MAKEFILE. Many different types of commands can be used within actions including conditionals, loops, constraints, and attribute assignments. Attributes of primitive implementation items on the right-hand-side may be accessed in a manner similar to Yacc: its position number on the right-hand-side followed by a "." and the attribute name. Special filters may be used to strip file extensions (e.g., `:r`).

Actions that occur between items on the right-hand side of a production are executed during the packaging process. The last action of a production rule, however, is special: it is only executed if the rule succeeds and is included in the software manufacture graph, i.e. during the traversal of the constructed manufacture graph. When the whole graph is built, the packager traverses the graph and executes these final actions.

The clause "[(OBJS) ...]" in the above rule specifies a synthesized attribute. Inherited attributes and constraints are scoped on the manufacture graph, but synthesized attributes must be explicitly stated at each rule. For example in the rule

```
cobjs :     cobj  cobjs
      [ (OBJS) $1(OBJS) $2(OBJS) ]
      {
      $1    # execute actions in the cobj subtree
      $2    # execute actions in the cobjs subtree
      }
      ;
```

the value of the OBJS attribute of both items is concatenated together to yield the list of names of all object files in a subtree of the manufacture graph.

Rule files are shared by all developers in an environment. It is the job of a system administrator to update rule files and keep them current. When a knew tool is purchased, its capabilities should be expressed in terms of object types and additional rules. A full rule file is shown in Appendix 1. A sample output MAKEFILE based on these rules for the factorial solution is shown in Appendix 2.

10 Related Work

We have presented the basic concepts of software packaging that build on prior concepts including software manufacturing and module interconnection languages (MIL). The next sections discuss related work in software manufacturing, heterogeneous and distributed systems, and configuration languages in general.

10.1 MAKE

The most well-known tool that automates the software integration process based on software manufacture graphs is the UNIX MAKE program [20]. Given a description of the dependencies between files in an application, MAKE invokes the tools needed to build an application. It determines the sequence in which tools are used to build an application. The software developer is responsible for specifying the dependencies between files and the steps needed to rebuild a file if one of its dependents is changed. These specification are stored in MAKEFILES. The partial order in a MAKEFILE specification is based on course-grain changes to files, i.e., if a file is updated, then dependent targets must be updated as well in order to maintain consistency. Other systems exist that base updates on more fine-grain changes, but the basic principle of such tools is to maintain an invariant condition on the program configuration.

The UNIX MAKE program allows designers to specify software manufacture graphs in order to automate the rebuilding of products should one or more of its components change. Dependencies in a MAKEFILE are specified as relationships between *targets* and *dependents*. Both targets and dependents correspond to files. Targets may be dependents of other targets. This establishes the partial order between components. Each command in a MAKEFILE is a list of commands (one per line) for rebuilding the target from the dependents. If one or more of the dependents change (i.e., its file date and time is later than the target), then the commands are executed. If the target is a dependent of another target, then execution of commands continues according to the partial order specified by the entire MAKEFILE.

In a MAKEFILE, the lines

```
.c.o:
      cc -c $*.c
```

specify a *suffix rule* that specifies that any file named with the extension ".c" can be translated into a file with the same name with the extension ".o" by using the *cc* tool (the C compiler). Such rules are often implicitly defined by each execution environment. A suffix rule is abstract in the sense that it applies to all files of a particular type specified by a file extension, e.g., `.c`. Such rules are limited in their ability to express dependencies and the integration capabilities within an environment.

If MAKEFILE works in one environment, it may not work in other environments. One of the major problems with MAKE is portability. MAKE was originally designed to maintain computer programs, but it has been extensively used to port and build programs across execution environments. Errors are commonplace when porting software manually via MAKEFILES. There are many differences between programming environments: compilers, IPC mechanisms, file paths, and installation options. Developers are forced to modify MAKEFILES directly because of such differences and include implementation alternatives based on platform-specific features. Macros alleviate some portability problems, but they are statically declared and globally scoped on the MAKEFILE and complex to use in large applications. Suffix rules also help because they are defined by the local environment, but such rules are limited to simple dependencies.

10.2 IMAKE

A better approach to portability is promoted by the IMAKE tool [21]. The IMAKE utility is a tool that handles portability problems by leaving it to the execution environment to define integration procedures. An IMAKEFILE is a portable configuration "program" that invokes these procedures. IMAKE is implemented using the C preprocessor that expands the procedure calls into MAKE production rules. For instance, the following is an IMAKEFILE that integrates our FORTRAN and C components:

Each execution environment defines its own procedures, i.e., in a *local.imake* file. This file contains implementations for integration procedures. Given these procedures and an IMAKEFILE, the IMAKE tool produces a MAKEFILE customized for the target environment. Although the implementation of IMAKE is via the C preprocessor, it allows integrations to be specified independent of target execution environments. It has been used successfully in the distribution of the X window system [22] across many execution environments. While MAKE dependency rules may be specified in any order, IMAKE procedures must be invoked in some sequence because some procedures depend on macros or rules declared by other procedures.

File inclusion mechanisms within MAKEFILES and IMAKEFILES provide a means to specify platform-independent configurations. By assigning targets to predefined macros, a developer can include a common set of rules as defined by the system. In the case of MAKE, standard macros are assigned values that parameterize predefined rules. This approach, however, is limited because multiple rules based on

lists of targets are not possible or limited. This is due to the fact that macro expansion is static and prevents the use of more complicated constructs such a list iterations and complex conditionals.

Like IMAKE, the software packager also relies on system-dependent integration "rules" but the major difference lies in the ability of the packager to infer integration steps rather than stating the procedures explicitly. IMAKE was developed for porting homogeneous software products between hardware platforms and does not easily handle heterogeneous integrations. It cannot infer the integration steps based on the types of components, but relies on the developer to invoke the proper integration procedures.

10.3 Program Changes

A major problem with using tools like MAKE and IMAKE is handling program evolution. If a component is changed, then MAKE can rebuild the target product. On the other hand, if the interface of a component changes or a component is added or removed, then the dependency relationships between components may change. In many cases, this means that the MAKEFILE must also be altered. Both MAKE and IMAKE were designed to maintain *static* dependency structures. They do not handle updating dependencies themselves that result from changing gross software structures. High-level configuration decisions can drastically change the nature of an integration process. For example, if we decide to reimplement the Driver component in C in the factorial example, then we can rewrite the MAKEFILE as

```
factorial: driver.o factorial.o
      cc -o a.out driver.o factorial.o

.c.o:
      cc -c $*.c
```

Similarly, new tools effect the types of possible integrations in an environment. If we introduce an C-to-FORTRAN translator (i.e., a hypothetical tool), we can configure the application to take advantage of this new capability by adding new production rules that characterize this new integration process. A new interprocess communications facility may also impact integration processes. Such changes to configurations and environments occur frequently in many environments because applications and their supporting platforms evolve with the introduction of new technologies.

One of the major advantages of software packaging is the ability to accommodate software reconfigurations without having to respecify the integration process for a product. This is a significant gain over existing methods where reconfigurations require drastic changes to integrations. We have used software packaging to produce MAKEFILE specifications automatically and show how small reconfiguration changes produce large changes to MAKEFILE specifications.

10.4 Other Manufacture Graph-Based Tools

Other tools that employ dependency-based methods for building software applications include NMAKE [23], GNUMAKE [24], SHAPE [25] and ODIN [26]. Some provide more sophisticated manipulation of macros and all employ file inclusion mechanisms or platform-dependent rules. All of these tools, however, require the explicit use of integration procedures that depend on the configuration characteristics of an application. The developer must specify the software manufacture graph explicitly in order to integrate an application. Mechanisms like suffix rules and IMAKE procedures automate the building process, but there is no relationship between rules and procedures in these systems.

We have seen that changes to the structure of an application or differences between execution environments can determine the integration process for an application. This means that the manufacture graph for an application can change from environment to environment. It can also change during

program development. Our approach relies on the use of an inference engine to derive manufacture graphs automatically. Like IMAKE, the software packager relies on each environment to specify its available integration processes. Our approach, however, differs because the rules describe the abstract integration processes not just the disjoint integration procedures available in an execution environment.

Suffix rules in MAKE, for example, are a simple forms of this abstraction. For example, the suffix rule

```
.c.o:
      cc -c $*.c
```

states that any file written in the C programming language can be compiled into an object file. The extension "`.c`" is a convention that identifies the file to be a certain *type* of component: a source file written in C. We have extended this notion to include complex relationships between types of components and the integration processes in execution environments.

10.5 Module Interconnection vs. Object-Oriented

The PACKAGE specification language is a module interconnection language with some unique features, namely, the ability to depict choices of implementations. Many configuration management tools present similar organizations of software structures using hierarchical file systems with enhancements for handling alternatives. For example, NMAKE depends on a standard directory structure for organizing product implementation alternatives. INTERCOL [27] presents a similar structure with implementation choices within the configuration language. Our approach is similar, but the choice of implementation for a component is not specified explicitly in the structure, rather it is left to the packager.

Many programming systems support the separation of interface and implementation to reduce coupling within programs. This separation allows designers to concentrate on distinct subparts of a problem. As long as the interface of a component remains fixed, its associated implementation is irrelevant to another developer using the interface. This allows programmers to work independently of one another and isolate changes to implementations. The separation greatly reduces coupling within programs. Coupling increases the likelihood that small changes will propagate extensively within a program. This increases the chance for errors and inconsistencies if done manually.

Object-oriented programming promotes the separation of interface and implementation, but it presents a single-implementation model. An interface specification for an object lists methods defined by the object. An interface is relatively independent of its implementation. Exceptions are usually for pragmatic reasons like performance (e.g., member function implementations and private variables defined within C++ class definitions). Most object-oriented systems are homogeneous; all components are implemented in the same language and executables are designed to execute within a single address space. Furthermore, there are no ``choices" because each interface has a single associated implementation. This is sufficient in a homogeneous environment, but lacks extensibility to heterogeneous configurations.

Another major difference between object-oriented and module-oriente programming involves the use of indirection. Traditionally, an interface defines a set of resources (e.g., methods) defined by an object. References to other objects are embedded within object implementations. This is sufficient because all objects have similar implementations (i.e., programming language, runtime support). A module interface, however, describes the resources defined and used by a module. References to other modules are always indirect [28]. This strictly encapsulates a software component. No implicit form of coupling is possible because all interactions are explicitly specified through the module interface. A module describes a software component as a self-contained entity [27]. Module-oriented programming subsumes object-oriented programming because the correspondence between a module interface and its implementations is one-to-many. Module-oriented programming allows developers to explicitly address interconnections between providers and users of resources. This additional level of indirection permits the rebinding of clients to services that were never intended to be used together thus enhancing software reuse.

10.6 Remote Procedure Call

Several projects have attempted to solve the problem of integrating heterogeneous, distributed applications through the use of remote procedure call (RPC). This technology is important to solve the mechanics of the integration problem, but it does not solve the larger problem of simplifying the integration process. Indeed, the software packager relies on stub generator tools to bridge applications.

The HORUS system [29] helps generate RPC stubs for many programming languages and communication protocols without reimplementing the stub generator in each execution environment. HORUS consists of a driver program that employs two schema files to achieve system independence: a machine-dependent schema and a language schema. The stub specification is given as input and HORUS produces stubs based on the features of the target machine and programming language. HORUS is a generic stub generator but the developer is responsible for using it and writing the stub specifications.

The HRPC project [30] takes a similar approach to stub generation as HORUS, but also employs runtime mechanisms to resolve differences between RPC protocols. Applications in a distributed system may employ different RPC protocols (i.e., Sun, NIDL, Courier). HRPC applications may connect to any of these services by dynamically determining which protocol to use at runtime. The stubs are not statically generated, but dynamically configured. This has performance implications, but once connections are established, the HRPC system is fixed until some change in the service occurs. Once again, the developer must write the stub specifications in the HRPC language, invoke stub generation tools, and link the HRPC library into the application.

Another project related to RPC is the Common Object Request Broker Architecture (CORBA) [31] under the direction of the Object Management Group (OMG) --- a consortium of vendors trying to standardize software components and the design of bridges between them to enable easier integration. CORBA provides for more complex interactions between programs than procedure calls, but like HORUS it defines an Interface Description Language (IDL) that is language and system independent. In general, the runtime design of CORBA is closely related to the notion of a *software bus* as discussed in the next section.

In general, RPC tools are necessary to bridge heterogeneous, distributed applications, but they do not make programming such applications easier. Developers must determine what tools to use. If the configuration of an application changes, the integration may change drastically. The developer must reintegrate the application by applying different tools. The software packager eliminates this step by determining the integration process automatically based on the types of components. It may employ RPC mechanisms as described in this section. If a configuration changes, the developer simply repackages the application.

10.7 The Software Bus

A improvement on remote procedure call involves adding a level of indirection between components in a heterogeneous, distributed system. Instead of components being directly connected to one another as client and server, a third-party process routes messages from one process to another. If a process produces a message on a port, the router directs the message to receiver(s) according to some mapping that may be statically or dynamically specified. This module-based approach is more flexible because participants in the system may come and go. For instance, in the middle of a session, the server process may be replaced with no affect on other processes. The router might queue impending messages to the server while a new server enrolls in the overall configuration.

This model of integration, known as the *software bus* model, views software components as pluggable modules into a communications backplane. It provides a great deal of flexibility in distributed,

heterogeneous environments by adding a level of indirection to transactions between runtime components. The bus routes messages between participants and queues undelivered messages for future delivery. This approach can implement remote procedure call as well as asynchronous interactions styles.

If there exists a runtime environment in which all the primitive components can operate, then we can build and execute the application. If we view each connection as a message channel, then the common runtime environment would include a communication mechanism to realize these channels. The task of the software packager is to choose compatible implementations of modules and derive a viable runtime environment that supports the execution of all constituent modules and their interconnections. The software packager determines whether or not an appropriate software bus exists based on the types of components and the integration tools in the target environment. In the process of integrating the components, they may need to be adapted and additional components such as wrappers and stubs may be introduced.

Several projects including the Portable Common Toolkit Environment (PCTE) [32], Polylith [15], PVM [33], CONIC [34], CORBA [31], Field [35], and the Portable Common Runtime (PCR) environment [36] are based on the software bus approach to software integration as a means of encapsulating software and promoting reuse.

11 Applications

We have used the software packager in many target execution environments to solve problems of heterogeneity and distribution when porting or configuring an application. The following is a list of possible application areas for software packaging. Details of each area can be found in [18]. Our approach is extensible to many areas of integration problems in computing systems. We summarize these uses and some additional application areas for this technology.

11.1 Portability

We have been very successful in using the software packager to create customized makefiles for different programs including a PACKAGE specification for the packager itself. Each environment supplies a rule file for the software packager to generate platform-specific integration programs (e.g., MAKEFILES). Furthermore, by including alternate implementations of components and structuring an application so that only a few components must be reimplemented, developers can minimize the cost of porting applications between environments. Software packaging allows developers to organize applications in this fashion independent from the way an application is integrated. Current methods do not separate these tasks and embed the structure of a program within the integration steps.

11.2 Heterogeneity

We have used the software packager to integrate program components written in different languages with compatible interfaces including C, C++, Ada, Tk/Tcl, Lisp, the UNIX shell, and FORTRAN. By describing these components as modules and connecting ports of modules instances, its has allowed our students to evolve an application by substituting new implementations for components and repackaging the application. Often, they are unaware of the integration stubs that are generated because they deal only with the logical architecture of the application when making any changes to an application.

11.3 Distribution

Like the heterogeneous components, distributed components that execute on different physical processors at runtime in an application need bridges to connect with one another. The execution location is simply viewed as an attribute assigned to a module instance. If the locations are different for components, this restricts the packager to create an integration solution based on a distributed communication mechanism.

We have built many distributed applications using the software packager that rely on different communication mechanisms to integrate programs including SunRPC and Polylith. The capability of SunRPC and Polylith stub generator tools are described in software packager rules and leveraged automatically in a target environment if needed to integrate an application. Future plans call for the incorporation of other distributed programming mechanisms into our packager rules for CORBA, Field, and NIDL stub generators.

11.4 Parallel Programming

We also use the software packager to describe parallel applications in which instances of modules represent separate tasks both in distributed environments (e.g., via Polylith) and for lightweight processing (e.g., Ada tasks). In these cases, asynchronous source and sink ports are parts of module interface descriptions. Future plans include the ability to incorporate RAPIDE [37] implementations for components to build discrete event simulations with a rich set of attributed module port types.

11.5 Configuration Management

We associate a VERSION attribute with each primitive implementation to classify different versions of code as alternate implementations. Constraints within a package specification and production rules may restrict the choices of implementations to this with the same version. This allows us to associate a distribution name with many different component versions and not have to keep track of which component versions are needed for a particular distribution of the software. The distribution name is an attribute assigned in the Root composite implementation. This can be changed in the PACKAGE specification for an application and then repackaged to produce a MAKEFILE specific for that distribution release of the application.

11.6 Genericity

So far, the target object type has been an executable object called `exec` (see Appendix 1), but we are not limited to this situation. Libraries, documents, and databases can be packaged as well. For example, we have package rules for libraries and DVI files. In addition, we can package an existing executable with input and output files to construct a single test in a test suite. Different "implementations" of the input and output components represent different test cases. Constraints on the input and output ensure that the appropriate cases are matched together.

12 Future Directions

Our experience with the software packager has shown that it is a useful tool that elides the integration problems in heterogeneous, distributed environments. It is extensible so that future integration tools can be leverage when developed. We have plans to explore many avenues of integration gives this tool. We briefly summarize these areas.

Incremental integration. When a component is reimplemented, the entire package must be regenerated and the integration must be build from the primitive implementations. An incremental approach is needed to reduce the costs of repackaging an application.

Graphical specification. Developers who have used PACKAGE specifications have remarked that it is useful to visualize the structure graph when designing their applications, but the textual language does not provide the appropriate views. Several graphical tools have been developed to construct structure graphs through direct manipulation but none of these has been satisfactory to date. The reasons for this is

primarily that the structure graph formalism itself was under evolution. A new graphical tool is needed to provide the visualization required by programmers.

Portability metrics. Since well-structured applications are modularized in such a way that reimplementation of a component has a low impact on the system as a whole, we suggest that portability can be quantified through measurements on software structure graphs. Such metrics would support claims of portability with values and give programmers guidelines during development regarding their design.

13 Conclusions

Programming-in-the-large has long been a vision of many programmers who have wished to reuse existing software components by combining them together in a modular fashion, but are stopped by the barriers of heterogeneity. Configuration programming languages and IPC mechanisms allow developers to combine existing software into new applications at a modular level, but they rely on the programmer's knowledge of the component types and the capabilities of integration tools. The result is that the integration itself is a complex programming task that is just as difficult as programming-in-the-small.

Software packaging promotes the view of a software application as a modular collection of subsystems and alternate implementations. This view is practically applicable and in agreement with current experiences of software developers. Most software products that are portable and configurable in heterogeneous, distributed environments have multiple implementations and are structured in a modular fashion to isolate dependencies with the design.

It is difficult to show that a programming language is abstract insofar that convenience to the programmer is increased. Convenience is defined loosely in terms of how "terse" it is for the programmer to specify a solution. The best we can hope to do is show that a new approach is more convenient than existing methods. Software packaging reduces the amount of work programmers must do to integrate applications in heterogeneous, distributed environments. By dealing with connections abstractly and using software packaging to infer how to implement the connections and select compatible implementations, the software developer reduces the amount of work necessary and reuses the bridges built by other developers. Software packaging also allows disparate applications to be composed and speeds the synthesis of new applications. Previous methods provided much of the integration technology, but left the programmer to specify the use of these tools explicitly. Software packaging relates such tools to the integration processes in an environment.

Software packaging embraces diversity and allows developers focus on composing different programs while ignoring incompatibilities that exist between them. This is important when changing the configuration of an application: porting it to new platforms, adding new implementations of components or features, and distributing it across a network of computers. This transparency, previously available only in homogeneous software development environments, is of most value when prototyping new applications from existing software. Often, prototype applications consist of existing programs that have been "patched" together. If the prototype is viable, components may be reimplemented so that they are more tightly bound together in a runtime configuration.

Finally, software packaging represents an extensible framework for software integration. It does not favor any particular environment, protocol, or programming language. Such mechanisms must be available, however, in order to integrate any application. This permits specialized tools to coexist and bridges to be built when more general standards do not exist. As new standards and tools are developed, the packager rules can be updated and existing applications can be repackaged.

In many environments, it is difficult to integrate heterogeneous, distributed software not because we lack the technology to do so but because the integration process is complex. An application may be a

patchwork of connections between different systems. If one component changes (i.e., is reimplemented or moved to another hardware platform), this has profound impact on the runtime organization of the entire system.

The software packager is independent of the particular technology used to integrate applications. It does, however, coordinate the use of these technologies and associated tools. The packager determines which tools are necessary based on a description of their integration characteristics. It offers seamless integration to developers of software systems whose components are programmed in different languages, are distributed, or whose application must be ported to many execution environments. It is independent of any specific integration technology and extensible to include new technologies. Our use of this approach in our research has many current applications and possible future directions.

References

[1] J. Callahan and J. Purtilo, A packaging system for heterogeneous execution environments, *IEEE Transactions on Software Engineering*, June 1991, Volume 17, Number 6, pp. 626-635.

[2] Strassmann, P., The Future Strategy of DoD's Computer Czar, Washington Technology, July 30, 1992, pp. 21-28.

[3] Feldman, S., D. Gay, M. Maimone and N. Schryer, A FORTRAN-to-C Converter, AT&T Bell Laboratories Technical Report 149, Murray Hill, NJ, October 1991.

[4] White, J. E., A High-Level Framework for Network-Based Resource Sharing, AFIPS National Computer Conference, 1976, pp. 561-570.

[5] Batory, D., J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise, GENESIS: An Extensible Database Management System, *IEEE Transactions on Software Engineering*, November 1988, Volume 14, Number 11, pp. 1711-1730.

[6] Swartout, W. and R. Balzer, On the Inevitable Intertwining of Specification and Implementation, Communications of the ACM, July 1982, Volume 25, Number 7, pp. 438-440.

[7] Garlan, D., G. Kaiser, D. Notkin, Using Tool Abstraction to Compose Systems, IEEE Computer, June 1992, Volume 25, Number 6, pp. 30-38.

[8] DeRemer, F. and H. Kron, Programming-in-the-large versus Programming-in-the-small, IEEE Transactions on Software Engineering, June 1976, Volume 2, Number 6, pp. 80-86.

[9] Harms, D. and B. Weide, Copying and Swapping: Influences on the Design of Reusable Software Components, IEEE Transactions on Software Engineering, May 1991, Volume 17, Number 5, pp. 424-435.

[10] Tichy, W., Software Development Control Based on System Structure Description, Carnegie Mellon University Computer Science Department, January 1980.

[11] Stroustrup, B., The C++ Programming Language, Addison-Wesley, Reading, MA, 1986.

[12] Magee, J. and J. Kramer, Constructing Distributed Systems in Conic, IEEE Transactions on Software Engineering, June 1989, Volume 15, Number 6, pp. 663-675.

[13] Borison, E., Program Changes and the Cost of Selective Recompilation, Carnegie Mellon University, July 1989.

[14] Sun Microsystems Computer Corp., Remote Procedure Call Protocol Specification, January 1985, Mountain View, CA.

[15] Purtilo, J., Polylith: An Environment to Support Management of Tool Interfaces, ACM SIGPLAN Symposium on Language Issues in Programming Environments, Seattle, WA, June 25-28 1985, pp. 12-18.

[16] Object Management Group Inc., Object Management Architecture Guide 1.0, Framingham, MA 01701, 492 Old Connecticut Path.

[17] Apollo Computer Inc., The Network Interface Definition Language, September 1983, Cupertino, CA.

[18] Callahan, J., Software Packaging, Ph.D. Dissertation. Published as University of Maryland Technical Report CS-TR-3093 and University of Maryland Institute for Advanced Computer Studies Report UMIACS-TR-93-56.

[19] Johnson, S., Yacc: Yet another compiler-compiler, Bell Laboratories, 1979.

[20] Feldman, S., Make: A Program for Maintaining Computer Programs, Software Practice and Experience, April 1979, Volume 9, Number 4, pp. 255-265.

[21] McNutt, D., Imake: Friend or Foe?, SunExpert, November 1991, Volume 2, Number 11, pp. 46-50.

[22] Wall, L., Configuring the X11 Window System, USENIX Summer 1987, July 1987, pp. 161-190.

[23] Fowler, G., The Fourth Generation Make, USENIX Conference --- Summer '85, June 1985, pp. 159-174.

[24] Smith, D., Make, O'Reilly and Associates, Sebastopol, CA, 1991.

[25] Mahler, A. and A. Lampen, An Integrated Toolset for Engineering Software Configurations, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, November 1988, pp. 191-200.

[26] Waite, W., V. Heuring, and U. Kastens, Configuration Control in Compiler Construction, International Workshop on Software Version and Configuration Control, January 1988, pp. 159-168.

[27] Tichy, W., Software Development Control Based on System Structure Description, Carnegie Mellon University Computer Science Department, January 1980.

[28] Wegner, P., Object Oriented Programming, OOPS Messenger, November 1990, Volume 1, Number 4, pp. 3-54.

[29] Gibbons, P., A Stub Generator for Multilanguage RPC in Heterogeneous Environments, IEEE Transactions on Software Engineering, year= January 1987, Volume 13, Number 1, pp. 77-87.

[30] Notkin, D. and A. Black and E. Lazowska and H. Levy and J. Sanislo and J. Zahorjan, Interconnecting Heterogeneous Computer Systems, Communications of the ACM, March 1988, Volume 31, Number 3, pp. 258-273.

[31] Object Management Group Inc., Object Management Architecture Guide 1.0, Framingham, MA 01701, 492 Old Connecticut Path.

[32] The Sema Group, The Software Bus --- its Objective: the Mutual Integration of Distributed Software Engineering Tools, May 1989, Trafalgar House, Richfield Ave., Reading, Berkshire RG18QA, UK.

[33] Geist, A., The Portable Virtual Machine (PVM) Environment, University of Tennessee Computer Science Department, April 1990, TR-1677.

[34] Magee, J. and J. Kramer, Constructing Distributed Systems in Conic, IEEE Transactions on Software Engineering, June 1989, Volume 15, Number 6, pp. 663-675.

[35] Reiss, S., Connecting Tools Using Message Passing in the Field Environment, IEEE Software, July 1990, Volume 7, Number 4, pp. 57-66.

[36] Weiser, M., PCR: The Portable Common Runtime Environment, Software Practice and Experience, May 1990, Volume 18, Number 5, pp. 112-130.

[37] Belz, F. and D. Luckham, A new approach to prototyping Ada-based hardware/software systems. In *Proceedings of the ACM Tri-Ada Conference*, Baltimore, MD, December 1990.

Appendix 1 A Sample Rule File

```
exec        :       cfmobj cobjs
            :{
            %#
            %# This Makefile produced automatically by
            %# Software Packager.  DO NOT EDIT.
            %#
            LIBDIR  = -L/usr/lib
            %FLIBS  = -lF77 -lI77 -lm
            %
            %$1(APPNAME): $1(OBJ) $2(OBJS) wrapper.o
            %       cc -o $1(APPNAME) $1(OBJ) $2(OBJS) wrapper.o $(LIBDIR) $$(FLIBS)
            %
            %wrapper.o: wrapper.c
            %       cc -c wrapper.c
            %
            %wrapper.c: Map
            %       f2cgen -o wrapper.c $1(NUM) -d $2(NUMS)
            %
            $1
            $2
            };
cfmobj      :       cfmain
            [ (APPNAME) $1(APPNAME) ]
            [ (NUM) $1(NUM) ]
            [ (OBJ) $1(FILE):r ".o " ]
            :{
            %$1(FILE):r.o: $1(FILE):r.c
            %       cc -c $1(FILE):r.c
            %
            $1
            };
cfmain      :       fmain
            [ (APPNAME) $1.APPNAME ]
            [ (NUM) $1.NUM ]
            [ (FILE) $1.FILE ]
            :{
            %$1.FILE:r.c: $1.FILE
            %       f2c $1.FILE
            %
```

```
            };
cobjs       :       cobj cobjs
            [ (NUMS) $1(NUM) " " $2(NUMS) ]
            [ (OBJS) $1(OBJ) $2(OBJS) ]
            :{
            $1
            $2
            }
            |
            ;
cobj        :       cfunc
            [ (NUM) $1.NUM ]
            [ (OBJS) $1.FILE:r ".o " ]
            :{
            %$1.FILE:r.o: $1.FILE
            %       cc -c $1.FILE
            %
            };
```

Appendix 2 A Sample Output Makefile

```
#
# This Makefile produced automatically by
# Software Packager.  DO NOT EDIT.
#
FLIBS   = -lF77 -lI77 -lm

a.out: driver.o  factorial.o wrapper.o
       cc -o a.out driver.o  factorial.o  wrapper.o -L/usr/lib $(FLIBS)

wrapper.o: wrapper.c
       cc -c wrapper.c

wrapper.c: Map
       f2cgen -o wrapper.c 5 -d 7

driver.o: driver.c
       cc -c driver.c

driver.c: driver.f
       f2c driver.f

factorial.o: factorial.c
       cc -c factorial.c
```